

Exploring Prefix-Tree Adders using Excel Spreadsheets

Setting up an explorative learning environment

Kjell Jeppson

Department of Microtechnology and Nanoscience
Chalmers University of Technology
Gothenburg, Sweden
e-mail: jeppson@chalmers.se

Per Larsson-Edefors

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden
e-mail: perla@chalmers.se

Abstract— A learning environment based on Microsoft Excel spreadsheets is presented allowing for fast and systematic exploration of different implementations of integer adder designs. The spreadsheet properties are exploited to illustrate both layout and timing properties of an adder design. The usefulness of the learning environment is demonstrated by several different examples. We also describe how the spreadsheet representation can aid the designer at the VHDL level.

Keywords— student learning environment, prefix-tree adders, EDA tools, spreadsheets

I. INTRODUCTION

In this paper we will demonstrate a learning sequence where Microsoft Excel is used as the learning environment for facilitating student understanding of basic concepts in integer adder design. There are two reasons why spreadsheets were adopted as the exploration tool for adder design: First, spreadsheets is a widely available software tool that integrates simple computation with spatial visualizations and, second, the structure of spreadsheets is similar to the computing structure of adders, where spreadsheet columns correspond to adder bit significance levels.

The main purpose of the presented learning sequence is to overcome the difficulties in understanding tree adder PG networks as they appear in text books [1]. One such example of a PG network representing the Brent-Kung prefix-tree adder is shown in Figure 1. Other examples of spreadsheet use as a learning tool can be found in [2-4].

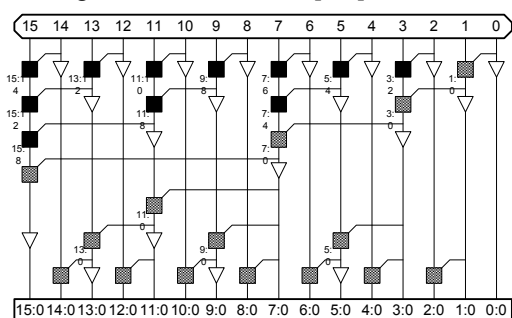


Figure 1. Brent-Kung tree adder PG network. From [1].

II. EXPLORING RIPPLE-CARRY ADDER STRUCTURES

The sequence of learning starts in the *Introduction to Integrated Circuit Design* course that runs September-October, with an exercise on designing an 8-bit ripple-carry adder. The exercise initially asks the students to use eight instances of the same bit-cell in an iterative logic array. Hence, the design task is nailed down into designing the carry and sum bit-cells. A bit-cell takes as inputs the two bits a_i and b_i and the carry-in, and delivers the carry-out and the

bit sum. The SUM logic is quite easy to define using two 2-input XOR stages, while the carry-in->carry-out logic requires more detailed considerations. At this stage, it is quite easy to check that the Boolean expressions are correct for the four different input combinations and the two different carry-in possibilities.

The carry logic expression can be written in Excel as

$$\text{=OR(AND(a;b);AND(cin;OR(a;b)))}$$

a logic expression corresponding to two AND-OR and OR-AND logic gates as shown in Figure 2, a figure that also shows the SUM XOR logic.

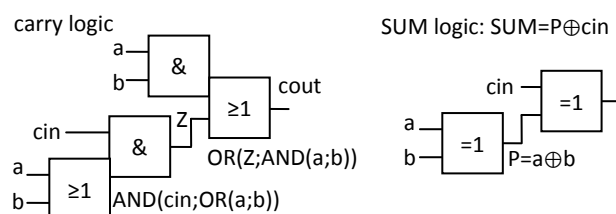


Figure 2. SUM and CARRY logic.

Once the bit-cell has been designed, the reuse of the same adder cell at several instances is easily demonstrated using the Excel click-and-drag facilities. Now, a ripple-carry adder has been designed and a possible layout structure is obvious as shown in the spreadsheet representation of Figure 3.

a7	b7	a6	b6	a5	b5	a4	b4	a3	b3	a2	b2	a1	b1	a0	b0	cin	
1	1	1	0	0	1	0	0	1	1	1	0	0	1	0	0	0	0 < cin
1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	PROP
0	0	1	1	1	0	0	0	0	1	1	1	1	1	0	0	0	SUM

Figure 3. Ripple-carry adder as iterative logic array.

Already now the student will be able to make a relatively good estimation of the area of the adder. This is because each row in the Excel graph could be seen as a row of standard cells from a standard-cell library. For a 65-nm CMOS process, the cell height (pitch) would be approximately 2.5 μm . The width of the AND-OR and OR-AND cells are approximately 1.5 μm , making the Excel column width corresponding to 3.0 μm in the adder layout.

From the sum logic expression in the ripple-carry adder, we can see that we need to perform the intermediate $P=a\oplus b$ operation as a preparation for calculating the sum. This operation need not wait for the rippling carry-in, but can be performed as a setup operation while waiting for the rippling carry to arrive, as shown in Figure 4. This design modification may not appear to be a big difference; however, as we will see in the following paragraphs, it will.

a7	b7	a6	b6	a5	b5	a4	b4	a3	b3	a2	b2	a1	b1	a0	b0		
1	1	1	0	0	1	0	0	1	1	1	0	0	1	0	0		
	0		1		1		0		0		1		1		0	P	
1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0 < cin	
	0		1		1		1		0		1		1		0	SUM	

Figure 4. Ripple-carry adder with propagate setup.

A closer study of the Boolean truth table for the ADD operation reveals that P is a signal indicating whether a previously generated carry should be propagated to the next bit or not. Hence, the carry logic can be simplified as shown in Figure 5.

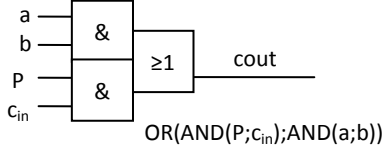


Figure 5. ANDOR22 ripple-carry logic.

The next step in developing an understanding of the ripple-carry design options, and preparing for an exploration of the prefix-tree adder design options, is to include also the generate logic, $G=ab$, into the setup logic. This is shown in Figure 6, and the corresponding simplified carry logic is shown in Figure 7. This redesign of the carry logic opts for a reduced footprint and for a shorter propagation delay through the ripple-carry logic as we will discuss in the next section.

a7	b7	a6	b6	a5	b5	a4	b4	a3	b3	a2	b2	a1	b1	a0	b0		
1	1	1	0	0	1	0	0	1	1	1	0	0	1	0	0		
1	0	0	1	0	1	0	0	1	0	0	1	0	0	0	0	G	P
1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0 < cin	
	0		1		1		1		0		1		1		0	SUM	

Figure 6. Ripple-carry adder with G and P setup logic.

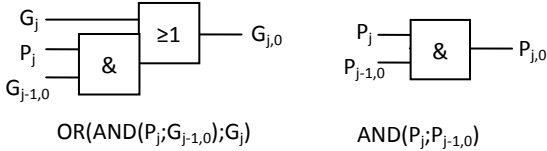


Figure 7. $G=AND-OR12$ and $P=AND2$ ripple-carry logic.

The ripple-carry logic shown in Figure 7 has many names; the dot operator or the prefix operator. The dot operator logic can be written,

$$\begin{cases} G_{i,j} = G_{i,k} + P_{i,k} G_{k-1,j} \\ P_{i,j} = P_{i,k} P_{k-1,j} \end{cases}$$

At this stage is appropriate to discuss with the students the associative properties of the dot operator. Once this property is understood by the student the carry-increment adder could be discussed. Since the dot operator is associative it does not matter in which order the dot operations are performed. This means that the carry-in need not be introduced in the beginning of a ripple chain, but could equally well be introduced at the end as shown in the carry-increment adder shown in Figure 8.

III. INTRODUCING THE TIME AXIS

The next performance parameter to study is the propagation delay from the clock edge enabling the carry-in until the

correct carry-out is available. At this stage of the *Introduction to Integrated Circuit Design* course very simple models for delay estimations are used. With short wires between cells, the propagation delay of any 2+1 AND-OR or OR-AND cell is dominated by the intrinsic delay, which is typically on the order of 50 ps per gate, i.e. 0.8 ns for the 8-bit carry-ripple chain.

a7	b7	a6	b6	a5	b5	a4	b4	a3	b3	a2	b2	a1	b1	a0	b0		
1	1	1	0	0	1	0	0	1	1	1	0	0	1	0	0		
1	0	0	1	0	1	0	0	1	0	0	1	0	0	0	0	G	P
G7	P7	G6	P6	G5	P5	G4	P4	G3	P3	G2	P2	G1	P1	G0	P0		
1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0		
G70	P70	G60	P60	G50	P50	G40	P40	G30	P30	G20	P20	G10	P10	G0	P0		
1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0 < cin	
	0		1		1		1		0		1		1		0	SUM	

Figure 8. Carry-increment adder.

The inherent propagation delay of the adder can be made more visible by including a time axis in the Excel chart as shown in Figure 9. Each row in the Excel work sheet now corresponds to a unit delay of $t(AO)=50$ ps.

a7	b7	a6	b6	a5	b5	a4	b4	a3	b3	a2	b2	a1	b1	a0	b0		
1	1	1	0	0	1	0	0	1	1	1	0	0	1	0	0		
1	0	0	1	0	1	0	0	1	0	0	1	0	0	0	0	G	P
G7	P7	G6	P6	G5	P5	G4	P4	G3	P3	G2	P2	G1	P1	G0	P0		
										G20	P20	0	0			t(AO)	
										0	0	G10	P10			2t(AO)	
																3t(AO)	
										0	0	G30	P30			4t(AO)	
										0	0	G40	P40			5t(AO)	
G70	P70	0	0	G50	P50											6t(AO)	
1	0	G60	P60													7t(AO)	
1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0 < cin	
	0		1		1		1		0		1		1		0	SUM	

Figure 9. Ripple-carry adder with delay information.

Any number of 8-bit ripple-carry blocks like the ones discussed in the previous section can now be used to build wide adders, like using four blocks for building a 32-bit adder. As an example, a 32-bit carry-skip adder can be organized as proposed in Figure 10 using the 8-bit ripple-carry propagate setup block from Figure 4 as the basic building block. In this adder structure, the skip multiplexer between the blocks lets the carry-in skip a block as soon as the block propagate is true.

The worst delay of this adder is obtained when a carry is generated in bit 0, i.e. the first bit of the least significant block. The carry then has to ripple through the first 8-bit block to reach the multiplexer, pass the multiplexer, skip the next two blocks via two more multiplexers, before arriving as a carry-in to the last block. Here, it must ripple through the block until it reaches the most significant bit where it serves as the carry-in. The 32-bit worst case propagation delay is then given by

$$t_{skip} = t_P + (7 + 7)t_{ripple} + 3t_{MUX} + t_{SUM},$$

where t_{ripple} is the ripple cell delay, t_{MUX} the multiplexer delay and t_{SUM} the delay of the final sum calculation. The general delay formula for an m -bit carry skip adder consisting of k n -bit blocks is then given by

$$t_{skip} = t_P + 2(n-1)t_{ripple} + (k-1)t_{MUX} + t_{SUM}.$$

Even more delay efficient is the 32-bit carry-lookahead adder (CLA) shown in Figure 11, an adder using the 8-bit block with G and P setup from Figure 6 as building block.

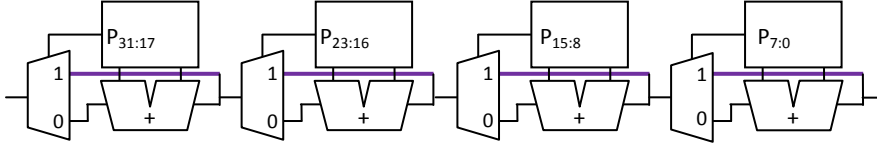


Figure 11. 32-bit carry-skip adder.

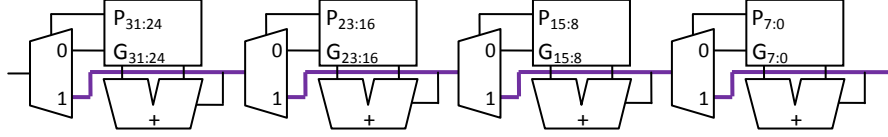


Figure 10. 32-bit carry-lookahead adder (CLA).

The propagation delay of the CLA is similar to that of the carry-skip adder, but with t_{MUX} and t_{ripple} replaced by the shorter t_{AO} , the propagation delay of an AND-OR12 gate:

$$t_{CLA} = t_{PG} + [2(n-1) + (k-1)]t_{AO} + t_{SUM}.$$

In the CLA design, a less complex AO gate can be used as multiplexer. Since P and G cannot both be true at the same time, the Boolean expression of the multiplexer can be simplified to

$$Z_{MUX} = \overline{P}G + PC_{in} = G + PC_{in}.$$

The instantiation of the simplified AND-OR multiplexer is shown marked in red in the previous Figures 6, 8, and 9.

The delay of an m -bit CLA built from k n -bit carry-increment block as shown in Figure 8 is now given by

$$t_{CLA} = t_{PG} + [n + k - 1]t_{AO} + t_{SUM}.$$

The difference between the two propagation delay formulas is illustrated in Figure 12.

bits 31:24	bits 23:16	bits 15:8	bits 7:0
t(PG) setup delay			
7-bit G & P ripple delay=7t(ripple)			
			t(MUX)
		t(MUX)	
7 t(ripple)			
t(SUM) propagation delay			
bits 31:24	bits 23:16	bits 15:8	bits 7:0
t(PG) setup delay			
7-bit G & P ripple delay=7t(AO)			
			t(AO)
		t(AO)	
t(AO)			
t(SUM) propagation delay			

Figure 12. Reducing propagation delay by carry-increment.

IV. EXPLORING PREFIX-TREE ADDERS

The block generate and propagate signals can be generated much more efficiently than by using ripple chains. Binary-tree structures can be used to reduce the block P and G delay considerably, especially for wide n -bit adder blocks. Outputs from a binary-tree structure are available at the block output already after $\log_2(n)$ unit delays. The spreadsheet representa-

tion of a CLA with a binary-tree P and G look-ahead design is shown in Figure 13. This CLA distributes the block P and G signals much faster to the subsequent 8-bit blocks, but it does not speed up the sum calculation within the different blocks. We will still have to wait for the carry to ripple to the most significant bit before all sums are calculated.

a7	b7	a6	b6	a5	b5	a4	b4	a3	b3	a2	b2	a1	b1	a0	b0		
1	1	1	0	0	1	0	0	1	1	1	0	0	1	0	0		
1	0	0	1	0	1	0	0	1	0	0	1	0	1	0	0	G	P
G7	P7	G6	P6	G5	P5	G4	P4	G3	P3	G2	P2	G1	P1	G0	P0		
1	0			0	0			1	0	G20	P20	0	0				t(AO)
1	0							1	0	0	0	G10	P10				2t(AO)
1	0							1	0								3t(AO)
						0	0	G30	P30								4t(AO)
				0	0	G40	P40										5t(AO)
G70	P70	0	0	G50	P50												6t(AO)
1	0	G60	P60														7t(AO)
1	0							1	0	0	0	0	0	0	0	0	<cin
0	0	1		1		1		0		1		1		0			SUM

Figure 13. Binary-tree carry-lookahead adder.

Now is the time for the students to start exploring the prefix-tree adder structures available in most text books. The knowledge acquired from designing ripple-carry adders using the logic functionality offered by Excel spreadsheets now serves as an excellent platform for exploring different prefix-tree adder structures. The spreadsheet learning environment gives instant feedback to the student whether the functionality of an adder design is correct or not.

A number of prefix-tree adder architectures have been published during the years; all with different trade-offs between propagation delay, wiring complexity, and number of prefix-tree dot operator cells needed. The most well-known prefix-tree adders are the Kogge-Stone, Han-Carlson, Sklansky, Ladner-Fischer, and Brent-Kung architectures. Already in the binary tree generating the block P and G outputs, the carries necessary for calculating half of the sums are available. The missing carries, and the corresponding missing sums, are shown by question marks in Figure 14. The problem facing the students now nails down to understanding how the binary tree can be extended to a complete prefix-tree supplying all necessary carries.

Sklansky used a brute-force carry-increment type of structure for obtaining the missing carries as illustrated in Figure 15. He solved the problem by adding a number of dot operator cells for distributing the missing carry in a way reminding of the previous carry-increment solution, but at many levels. The cells added are marked in blue. The logical depth of the tree is given by $\log_2(n)+1$, but there is a capacitive loading problem hidden in this structure for the student to identify and solve by inserting buffers.

a7	b7	a6	b6	a5	b5	a4	b4	a3	b3	a2	b2	a1	b1	a0	b0		
1	1	1	0	0	1	0	0	1	1	1	0	0	1	0	0		
1	0	0	1	0	1	0	0	1	0	0	1	0	1	0	0	G	P
G7	P7	G6	P6	G5	P5	G4	P4	G3	P3	G2	P2	G1	P1	G0	P0		
1	0			0	0			1	0			0	0				
1	0							1	0								
1	0															0 <cin	
1		?	?	?	?	1		?		0		0				0	
	?		?	?		1		?	1		1		1		0	SUM	

Figure 14. Missing carries in the P and G binary tree.

He solved the problem by adding a number of dot operator cells for distributing the missing carry in a way reminding of the previous carry-increment solution, but at many levels. The cells added are marked in blue. The logical depth of the tree is given by $\log_2(n)+1$, but there is a capacitive loading problem hidden in this structure for the student to identify and solve by inserting buffers.

a7	b7	a6	b6	a5	b5	a4	b4	a3	b3	a2	b2	a1	b1	a0	b0		
1	1	1	0	0	1	0	0	1	1	1	0	0	1	0	0		
1	0	0	1	0	1	0	0	1	0	0	1	0	1	0	0	G	P
G7	P7	G6	P6	G5	P5	G4	P4	G3	P3	G2	P2	G1	P1	G0	P0		
1	0			0	0			1	0			0	0				
1	0	0	0					1	0	0	0						
1	0	0	0	0	0	0	0									1 <cin	
1		0		0		0		1		0		0		0		1	
	0		1		1		1		0		1		1		1	SUM	

Figure 15. Sklansky prefix-tree adder.

Ladner-Fischer proposed a solution relieving some of the loading problem by redistributing the bottom row of dot operators into two levels as shown in Figure 16.

a7	b7	a6	b6	a5	b5	a4	b4	a3	b3	a2	b2	a1	b1	a0	b0		
1	1	1	0	0	1	0	0	1	1	1	0	0	1	0	0		
1	0	0	1	0	1	0	0	1	0	0	1	0	1	0	0	G	P
G7	P7	G6	P6	G5	P5	G4	P4	G3	P3	G2	P2	G1	P1	G0	P0		
1	0			0	0			1	0			0	0				
1	0							1	0								
1	0			0	0			1	0								
1				0				1				0				0 <cin	
		0				0				0				0			
	0		1		1		1		0		1		1		0	SUM	

Figure 16. Ladner-Fischer prefix-tree adder

Finally, Brent-Kung proposed a solution where they got away without adding cells to the prefix tree. Instead they reorganized the bottom row of cells into an inverse tree distributing the missing carries, see Figure 17.

V. DESIGN METHODOLOGY

Now, after a number of exercises where students learn the fundamentals of integrated CMOS circuits using the adder as the design example, the first learning stage is completed.

In the second learning stage, student focus is shifted to design methodology and EDA tools. In the *Methods of Electronic System Design and Verification* course running October-December, the design abstraction level is raised from MOSFETs and circuits to the RTL level. In this unique methodology-oriented course [5], students are set to learn about the capabilities and limitations of EDA tools for synthesis, timing closure, place and route, etc.

The task assigned to the students in this course, is the design, implementation and verification of a 32-bit ALU, using Cadence RTL Compiler and Encounter for a 65-nm low-power technology.

a7	b7	a6	b6	a5	b5	a4	b4	a3	b3	a2	b2	a1	b1	a0	b0		
1	1	1	0	0	1	0	0	1	1	1	0	0	1	0	0		
1	0	0	1	0	1	0	0	1	0	0	1	0	1	0	0	G	P
G7	P7	G6	P6	G5	P5	G4	P4	G3	P3	G2	P2	G1	P1	G0	P0		
1	0			0	0			1	0			0	0				
1	0							1	0								
1	0							1	0								
1								1								0 <cin	
		0				0				0		0		0			
		0		1		1		1		0		1		1		0	SUM

Figure 17. Brent-Kung prefix-tree adder.

Initially, the students implement a ripple-carry adder for the ALU, but since the target clock rate is 1 GHz, they run into timing problems as the ALU is synthesized. The knowledge about prefix-tree adders acquired by students in the previous course now comes in handy. By replacing the slow ripple-carry adder with a faster Sklansky adder, students are able to close the timing.

The dot operator code below is an excerpt of the VHDL skeleton that is available to the students. By matching the computational structure of the code to the Excel representation it becomes relatively straightforward to match the indices of the code to the significance levels of the adder.

```

row: for i in 1 to INTEGER(CEIL(LOG2(REAL(32)))) generate
column: for j in 0 to 31 generate
a: if
((INTEGER(FLOOR(REAL(j)/REAL(2**(i-1)))) mod 2)=1
generate
b: if j < 2**(i) generate
Gdotops: Gdot port map (
P1=>P(i-1)(j),
G1=>G(i-1)(j),
G2=>G(i-1)--!find correct description!,
G=>G(i)(j));
-- P does not need to be propagated
end generate b;

c: if j >= 2**(i) generate
Dotops: DOTs port map(
P1=>P(i-1)(j),
G1=>G(i-1)(j),
P2=>P(i-1)--!find correct description!,
G2=>G(i-1)--!find correct description!,
P=>P(i)(j),
G=>G(i)(j));
end generate c;
end generate a;

```

VI. SUMMARY AND CONCLUSION

In this paper we have illustrated the use of Excel spreadsheets as a student learning tool for understanding digital design of integer adder structures. The tool emphasizes important concepts such as layout through instantiation of identical cells by click-and-drag, timing constraints caused by cell propagation delay through introduction of a timing axis, and yields a proper understanding of prefix-adder trees.

VII. REFERENCES

- [1] N. Weste and D. Harris, Integrated Circuit Design, 4th ed., Boston: Pearson Education, 2011, ch. 10.2.
- [2] I. Levin and H. Rosensweig, Spreadsheet Learning Environment for Teaching Advanced Topics in Computer Engineering, EWME 2012, Grenoble, France, pp 45-47
- [3] T. Piontek and C. Osterloh, Notion of Time and Parallelism: What Computer Science Students need to know about Digital Circuit Design, EWME 2012, Grenoble, France, pp 48-49
- [4] I. Levin, Behavioral Simulation of an Arithmetic Unit using the Spreadsheet, Int. J. Engineering Education, Vol 31, pp. 334-341.
- [5] P. Larsson-Edefors and K. Jeppson, Training Design Methodology Skills at the Master's Level, EWME 2012, Grenoble, pp 10-13.